CYCLE COUNT REPLICATION IN A SIMULTANEOUS AND REDUNDANTLY THREADED PROCESSOR

CROSS-REFERENCE TO RELATED APPLICATIONS

[0001] This application is a non-provisional application claiming priority to provisional application Serial No. 60/198,530, filed on April 19, 2000, entitled "Transient Fault Detection Via Simultaneous Multithreading," the teachings of which are incorporated by reference herein.

[0002] This application is further related to the following co-pending applications, each of which is hereby incorporated herein by reference:

[0003] U.S. Patent Application No. _____, filed _____, and entitled "Slack Fetch to Improve Performance of a Simultaneous and Redundantly Threaded Processor," Attorney Docket No. 1662-23801;

[0004] U.S. Patent Application No. _____, filed _____, and entitled "Simultaneous and Redundantly Threaded Processor Store Instruction Comparator," Attorney Docket No. 1662-36900;

[0005] U.S. Patent Application No. _____, filed _____, and entitled "Active Load Address Buffer," Attorney Docket No. 1662-37100;

[0006] U.S. Patent Application No. _____, filed _____, and entitled "Simultaneous and Redundantly Threaded Processor Branch Outcome Queue," Attorney Docket No. 1662-37200;

[0007] U.S. Patent Application No. _____, filed _____, and entitled "Input Replicator for Interrupts in a Simultaneous and Redundantly Threaded Processor," Attorney Docket No. 1662-37300;

[0008] U.S. Patent Application No. _____, filed _____, and entitled "Simultaneous and Redundantly Threaded Processor Uncached Load Address Comparator and Data Value Replication Circuit," Attorney Docket No. 1662-37400;

[0009] U.S. Patent Application No. _____, filed _____, and entitled "Load Value Queue Input Replication in a Simultaneous and Redundantly Threaded Processor," Attorney Docket No. 1662-37500.

BACKGROUND OF THE INVENTION

Field of the Invention

[0010] The present invention generally relates to microprocessors. More particularly, the present invention relates to a pipelined, multithreaded processor that can execute a program in at least two separate, redundant threads. More particularly still, the invention relates to a method and apparatus for ensuring valid replication of reads from a cycle counter to each redundant thread.

Background of the Invention

[0011] Solid state electronics, such as microprocessors, are susceptible to transient hardware faults. For example, cosmic rays or alpha particles can alter the voltage levels that represent data values in microprocessors, which typically include millions of transistors. Cosmic radiation can change the state of individual transistors causing faulty operation. The frequency of such transient faults is relatively low—typically less than one fault per year per thousand computers. Because of

this relatively low failure rate, making computers fault tolerant currently is attractive more for mission-critical applications, such as online transaction processing and the space program, than computers used by average consumers. However, future microprocessors will be more prone to transient fault due to their smaller anticipated size, reduced voltage levels, higher transistor count, and reduced noise margins. Accordingly, even low-end personal computers may benefit from being able to protect against such faults.

[0012] One way to protect solid state electronics from faults resulting from cosmic radiation is to surround the potentially effected electronics by a sufficient amount of concrete. It has been calculated that the energy flux of the cosmic rays can be reduced to acceptable levels with six feet or more of concrete surrounding the computer containing the chips to be protected. For obvious reasons, protecting electronics from faults caused by cosmic ray with six feet of concrete usually is not feasible. Further, computers usually are placed in buildings that have already been constructed without this amount of concrete.

[0013] Rather than attempting to create an impenetrable barrier through which cosmic rays cannot pierce, it is generally more economically feasible and otherwise more desirable to provide the affected electronics with a way to detect and recover from a fault caused by cosmic radiation. In this manner, a cosmic ray may still impact the device and cause a fault, but the device or system in which the device resides can detect and recover from the fault. This disclosure focuses on enabling microprocessors (referred to throughout this disclosure simply as "processors") to recover from a fault condition. One technique, such as that implemented in the Compaq Himalaya system, includes two identical "lockstepped" microprocessors. Lockstepped processors have their clock cycles synchronized and both processors are provided with identical inputs (*i.e.*, the same instructions to execute, the same data, etc.). A checker circuit compares the processors' data output

(which may also include memory addressed for store instructions). The output data from the two processors should be identical because the processors are processing the same data using the same instructions, unless of course a fault exists. If an output data mismatch occurs, the checker circuit flags an error and initiates a software or hardware recovery sequence. Thus, if one processor has been affected by a transient fault, its output likely will differ from that of the other synchronized processor. Although lockstepped processors are generally satisfactory for creating a fault tolerant environment, implementing fault tolerance with two processors takes up valuable real estate.

[0014] A "pipelined" processor includes a series of functional units (*e.g.*, fetch unit, decode unit, execution units, etc.), arranged so that several units can be simultaneously processing an appropriate part of several instructions. Thus, while one instruction is being decoded, an earlier fetched instruction can be executed. A "simultaneous multithreaded" ("SMT") processor permits instructions from two or more different program threads (*e.g.*, applications) to be processed through the processor simultaneously. An "out-of-order" processor permits instructions to be processed in an order that is different than the order in which the instructions are provided in the program (referred to as "program order"). Out-of-order processing potentially increases the throughput efficiency of the processor. Accordingly, an SMT processor can process two programs simultaneously.

[0015] An SMT processor can be modified so that the same program is simultaneously executed in two separate threads to provide fault tolerance within a single processor. Such a processor is called a simultaneous and redundantly threaded ("SRT") processor. Some of the modifications to turn a SMT processor into an SRT processor are described in Provisional Application Serial No. 60/198,530.

[0016] Executing the same program in two different threads permits the processor to detect faults such as may be caused by cosmic radiation, noted above. By comparing the output data from the two threads at appropriate times and locations within the SRT processor, it is possible to detect whether a fault has occurred. For example, data written to cache memory or registers that should be identical from corresponding instructions in the two threads can be compared. If the output data matches, there is no fault. Alternatively, if there is a mismatch in the output data, a fault has presumably occurred in one or both of the threads.

[0017] Executing the same program in two separate threads advantageously affords the SRT processor some degree of fault tolerance, but also may cause several performance problems. For instance, any latency caused by a cache miss is exacerbated. Cache misses occur when an instruction requests data from memory that is not also available in cache memory. The processor first checks whether the requested data already resides in the faster access cache memory, which generally is onboard the processor die. If the requested data is not present in cache (a condition referred to as a cache "miss"), then the processor is forced to retrieve the data from main system memory which takes more time, thereby causing latency, than if the data could have been retrieved from the faster onboard cache. Because the two threads are executing the same instructions, any instruction in one thread that results in a cache miss will also experience the same cache miss when that same instruction is executed in other thread. That is, the cache latency will be present in both threads.

[0018] A second performance problem concerns branch misspeculation. A branch instruction requires program execution either to continue with the instruction immediately following the branch instruction if a certain condition is met, or branch to a different instruction if the particular condition is not met. Accordingly, the outcome of a branch instruction is not known until the

instruction is executed. In a pipelined architecture, a branch instruction (or any instruction for that matter) may not be executed for at least several, and perhaps many, clock cycles after the branch instruction is fetched by the fetch unit in the processor. In order to keep the pipeline full (which is desirable for efficient operation), a pipelined processor includes branch prediction logic which predicts the outcome of a branch instruction before it is actually executed (also referred to as "speculating"). Branch prediction logic generally bases its speculation on short or long term history. As such, using branch prediction logic, a processor's fetch unit can speculate the outcome of a branch instruction before it is actually executed. The speculation, however, may or may not turn out to be accurate. That is, the branch predictor logic may guess wrong regarding the direction of program execution following a branch instruction. If the speculation proves to have been accurate, which is determined when the branch instruction is executed by the processor, then the next instructions to be executed have already been fetched and are working their way through the pipeline.

[0019] If, however, the branch speculation turns out to have been the wrong prediction (referred to as "misspeculation"), many or all of the instructions filling the pipeline behind the branch instruction may have to be thrown out (*i.e.*, not executed) because they are not the correct instructions to be executed after the branch instruction. The result is a substantial performance hit as the fetch unit must fetch the correct instructions to be processed through the pipeline. Suitable branch prediction methods, however, result in correct speculations more often than misspeculations and the overall performance of the processor is improved with a suitable branch predictor (even in the face of some misspeculations) than if no speculation was available at all.

[0020] In an SRT processor that executes the same program in two different threads for fault tolerance, any branch misspeculation is exacerbated because both threads will experience the same

misspeculation. Because the branch misspeculation occurs in both threads, the processor's internal resources usable to each thread are wasted while the wrong instructions are replaced with the correct instructions.

[0021] In an SRT processor, threads may be separated by a predetermined amount of slack to improve performance. In this scenario, one thread is processed ahead of the other thread thereby creating a "slack" of instructions between the two threads so that the instructions in one thread are processed through the processor's pipeline ahead of the corresponding instructions from the other thread. The thread whose instructions are processed earlier is called the "leading" thread, while the other thread is the "trailing" thread. By setting the amount of slack (in terms of numbers of instructions) appropriately, all or at least some of the cache misses or branch misspeculations encountered by the leading thread can be resolved before the corresponding instructions from the trailing thread are fetched and processed through the pipeline.

[0022] In an SRT processor, the processor verifies that inputs to the multiple threads are identical to guarantee that both execution copies or threads follow precisely the same path. Thus, corresponding operations that input data from other locations within the system (*e.g.*, memory, cycle counter), must return the same data values to both redundant threads. Otherwise, the threads may follow divergent execution paths, leading to different outputs that will be detected and handled as if a hardware fault occurred.

[0023] One potential problem in running two separate, but redundant threads in a computer processor arises in reading the current value in the system cycle counter. A cycle counter is a running counter that advances once for each tick of the processor clock. Thus, for a 1 GHz processor, the counter will advance once every nanosecond. A conventional cycle counter may be

a 64-bit counter that counts up from zero to the maximum value and wraps around to zero to continue counting.

[0024] A program that is running on the processor may periodically request the current value of the cycle counter using a read or fetch command. For example, Compaq Alpha servers execute an "rpcc" command that is included in the instruction set for Alpha processors. By reading the cycle counter at the start and finish of an instruction or set of instructions, the processor may calculate how many clock cycles (and therefore, how much time) elapsed during execution of the instructions. Thus, the "read cycle counter" command provides a means of measuring system performance.

[0025] As discussed above, corresponding instructions in redundant threads are not executed at precisely the same time. Thus, it should be expected that corresponding read cycle count commands from the different threads will always return different values because some amount of time will elapse between the cycle count retrievals. While this cycle count variation between threads may be expected, the different values may result in a fault condition because the inputs to the two threads are different. It is desirable therefore, to develop a method of replicating the cycle count values from the cycle counter for each redundant thread in the pipeline. By replicating the cycle counter value, erroneous transient fault conditions or faulty SRT operation resulting from the trailing "read cycle count" instructions are avoided.

BRIEF SUMMARY OF THE INVENTION

[0026] The problems noted above are solved in large part by a simultaneous and redundantly threaded processor that can simultaneously execute the same program in two separate threads to provide fault tolerance. By simultaneously executing the same program twice, the system can be

made fault tolerant by checking the output data pertaining to corresponding instructions in the threads to ensure that the data matches. A data mismatch indicates a fault in the processor effecting one or both of the threads. The preferred embodiment of the invention provides an increase in performance to such a fault tolerant, simultaneous and redundantly threaded processor.

[0027] The preferred embodiment includes a pipelined, simultaneous and redundantly threaded ("SRT") processor, comprising a program counter configured to assign program count identifiers to instructions in each thread, a register update unit configured to store a queue of instructions prior to execution by the processor, load/store units configured to perform load and store operations to or from data locations such as a data cache and data registers, and a cycle counter configured to keep a running count of processor clock cycles. The processor is configured to detect transient faults during program execution by executing instructions in at least two redundant copies of a program thread. False errors caused by incorrectly replicating cycle count values in the redundant program threads are avoided by using the actual values from cycle count reads in a first program thread for the second program thread. The SRT processor is an out-of-order processor capable of executing instructions in the most efficient order, but read cycle count ("RCC") instructions are executed in the same order in both the first and second program threads. The register update unit is capable of managing program order for the RCC instructions by establishing a dependence with instructions before and after the RCC instructions in the register update unit.

[0028] The SRT processor further comprises a cycle count queue for storing the actual values fetched by RCC instructions in the first program thread. The load/store units place a duplicate copy of the cycle count value in the cycle count queue after fetching the cycle count value from the cycle counter. The load/store units then access the cycle count queue, and not the cycle counter, to fetch cycle count values in response to corresponding RCC instructions in the second program

thread. The cycle count queue is preferably a FIFO buffer and individual cycle count entries stored in the cycle count queue comprise: a program count assigned to the RCC instruction by the program counter and a cycle count value that was returned by the corresponding RCC instruction in the leading thread. If the cycle count queue becomes full, the first thread is stalled to prevent more cycle count values from entering the cycle count queue. Conversely, if the cycle count queue becomes empty, the second thread may be stalled to allow cycle count values to enter the cycle count queue.

[0029] An alternative embodiment exists for use in systems that do not have access to a cycle count queue. In this embodiment, the processor executes the redundant threads with some predetermined amount of slack between the threads. Upon encountering an RCC command in the leading thread, the leading thread is halted and the trailing thread is executed until the corresponding RCC command is reached in the trailing thread. Once synchronized, the load/store units fetch the current cycle count value from the cycle counter and distributes this value to both threads. The alternative embodiment permits implementation in existing computer systems.

BRIEF DESCRIPTION OF THE DRAWINGS

[0030] For a detailed description of the preferred embodiments of the invention, reference will now be made to the accompanying drawings in which:

[0031] Figure 1 is a diagram of a computer system constructed in accordance with the preferred embodiment of the invention and including a simultaneous and redundantly threaded processor;

[0032] Figure 2 is a graphical depiction of the input replication and output comparison executed by the simultaneous and redundantly threaded processor according to the preferred embodiment;

[0033] Figure 3 conceptually illustrates the problem encountered by the multithreaded processor of Figures 1 and 2 when corresponding cycle count read commands are issued at different cycle count values;

[0034] Figure 4 is a block diagram of the simultaneous and redundantly threaded processor from Figure 1 in accordance with the preferred embodiment that includes a single cycle counter and a cycle count queue;

[0035] Figure 5 is a diagram of a Register Update Unit in accordance with a preferred embodiment; and

[0036] Figure 6 is a diagram of a Cycle Count Queue in accordance with a preferred embodiment.

NOTATION AND NOMENCLATURE

[0037] Certain terms are used throughout the following description and claims to refer to particular system components. As one skilled in the art will appreciate, microprocessor companies may refer to a component by different names. This document does not intend to distinguish between components that differ in name but not function. In the following discussion and in the claims, the terms "including" and "comprising" are used in an open-ended fashion, and thus should be interpreted to mean "including, but not limited to...". Also, the term "couple" or "couples" is intended to mean either an indirect or direct electrical connection. Thus, if a first device couples to a second device, that connection may be through a direct electrical connection, or through an indirect electrical connection via other devices and connections.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

[0038] Figure 1 shows a computer system 90 including a pipelined, simultaneous and redundantly threaded ("SRT") processor 100 constructed in accordance with the preferred embodiment of the invention. In addition to processor 100, computer system 90 also includes dynamic random access memory ("DRAM") 92, an input/output ("I/O") controller 93, and various I/O devices which may include a floppy drive 94, a hard drive 95, a keyboard 96, and the like. The I/O controller 93 provides an interface between processor 100 and the various I/O devices 94-96. The DRAM 92 can be any suitable type of memory devices such as RAMBUS™ memory. In addition, SRT processor 100 may also be coupled to other SRT processors if desired in a commonly known "Manhattan" grid, or other suitable architecture.

[0039] The preferred embodiment of the invention ensures correct operation and provides a performance enhancement to SRT processors. The preferred SRT processor 100 described above is capable of processing instructions from two different threads simultaneously. Such a processor in fact can be made to execute the same program as two different threads. In other words, the two threads contain the same program set. Processing the same program through the processor in two different threads permits the processor to detect faults caused by cosmic radiation or alpha particles as noted above.

[0040] Figure 2 conceptually shows the simultaneous and redundant execution of threads 250, 260 in the processor 100. The threads 250, 260 are referred to as Thread 0 ("T0") and Thread 1 ("T1"). In accordance with the preferred embodiment, the processor 100 or a significant portion thereof resides in a sphere of replication 200, which defines the boundary within which all activity and states are replicated either logically or physically. Values that cross the boundary of the sphere of replication are the outputs and inputs that require comparison 210 and replication

220, respectively. Thus, a sphere of replication 200 that includes fewer components may require fewer replications but may also require more output comparisons because more information crosses the boundary of the sphere of replication. The preferred sphere of replication is described in conjunction with the discussion of Figure 4 below.

[0041] All inputs to the sphere of replication 200 must be replicated 220. For instance, an input resulting from a memory load command must return the same value to each execution thread 250, 260. If two distinctly different values are returned, the threads 250, 260 may follow divergent execution paths. Similarly, the outputs of both threads 250, 260 must be compared 210 before the values contained therein are shared with the rest of the system 230. For instance, each thread may need to write data to memory 92 or send a command to the I/O controller 93. If the outputs from the threads 250, 260 are identical, then it is assumed that no transient faults have occurred and a single output is forwarded to the appropriate destination and thread execution continues. Conversely, if the outputs do not match, then appropriate error recovery techniques may be implemented to re-execute and re-verify the "faulty" threads.

[0042] It should be noted that the rest of the system 230, which may include such components as memory 92, I/O devices 93-96, and the operating system need not be aware that two threads of each program are executed by the processor 100. In fact, the preferred embodiment generally assumes that all input and output values or commands are transmitted as if only a single thread exists. It is only within the sphere of replication 200 that the input or output data is replicated.

[0043] Among the inputs that must be replicated for distribution to the execution threads 250, 260 are cycle counter values that are periodically requested by computer programs. Figure 3 illustratively shows the problem with running two separate threads with corresponding "read cycle count" ("RCC") instructions. Figure 3 shows two distinct, but replicated copies of a program

thread T0 & T1 presumably executed in the same pipeline. Thread T0 is arbitrarily designated as the "leading" thread while thread T1 is designated as the "trailing" thread. The threads may be separated in time by a pre-determined slack and may also be executed out of program order. In the example shown in Figure 3, an RCC command is issued in the leading thread T0 that returns a cycle count value of "4". Because of the time delay between execution threads, the corresponding RCC command in trailing thread T1 is not issued until clock cycle "19". While this condition is perfectly normal and expected, the unequal inputs unfortunately yield a fault condition because the inputs to the sphere of replication 200 are not identical. This condition may be rectified by implementing the SRT processor 100 shown in Figure 4.

[0044] Referring to Figure 4, processor 100 preferably comprises a pipelined architecture which includes a series of functional units, arranged so that several units can be simultaneously processing appropriate parts of several instructions. As shown, the exemplary embodiment of processor 100 includes a fetch unit 102, one or more program counters 106, an instruction cache 110, decode logic 114, register rename logic 118, floating point and integer registers 122, 126, a register update unit 130, execution units 134, 138, and 142, a data cache 146, a cycle counter 148 and a cycle count queue 150.

[0045] Fetch unit 102 uses a program counter 106 for assistance as to which instruction to fetch. Being a multithreaded processor, the fetch unit 102 preferably can simultaneously fetch instructions from multiple threads. A separate program counter 106 is associated with each thread. Each program counter 106 is a register that contains the address of the next instruction to be fetched from the corresponding thread by the fetch unit 102. Figure 4 shows two program counters 106 to permit the simultaneous fetching of instructions from two threads. It should be recognized,

however, that additional program counters can be provided to fetch instructions from more than two threads simultaneously.

[0046] As shown, fetch unit 102 includes branch prediction logic 103 and a "slack" counter 104. Slack counter 104 is used to create a delay of a desired number of instructions between the threads that include the same instruction set. The introduction of slack permits the leading thread T0 to resolve all or most branch misspeculations and cache misses so that the corresponding instructions in the trailing thread T1 will not experience the same latency problems. The branch prediction logic 104 permits the fetch unit 102 to speculate ahead on branch instructions as noted above. In order to keep the pipeline full (which is desirable for efficient operation), the branch predictor logic 103 speculates the outcome of a branch instruction before the branch instruction is actually executed. Branch predictor 103 generally bases its speculation on previous instructions. Any suitable speculation algorithm can be used in branch predictor 103.

[0047] Referring still to Figure 4, instruction cache 110 provides a temporary storage buffer for the instructions to be executed. Decode logic 114 retrieves the instructions from instruction cache 110 and determines the instruction type (*e.g.*, add, subtract, load, store, etc.). Decoded instructions are then passed to the register rename logic 118 which maps logical registers onto a pool of physical registers.

[0048] The register update unit ("RUU") 130 provides an instruction queue for the instructions to be executed. The RUU 130 serves as a combination of global reservation station pool, rename register file, and reorder buffer. The RUU 130 breaks load and store instructions into an address portion and a memory (*i.e.*, register) reference. The address portion is placed in the RUU 130, while the memory reference portion is placed into a load/store queue (not specifically shown in Figure 4).

[0049] The RUU 130 also handles out-of-order execution management. As instructions are placed in the RUU 130, any dependence between instructions (*e.g.*, one instruction depends on the output from another or because branch instructions must be executed in program order) is maintained by placing appropriate dependent instruction numbers in a field associated with each entry in the RUU 130. Figure 5 provides a simplified representation of the various fields that exist for each entry in the RUU 130. Each instruction in the RUU 130 includes an instruction number, the instruction to be performed, and a dependent instruction number ("DIN") field. As instructions are executed by the execution units 134, 138, 142, dependency between instructions can be maintained by first checking the DIN field for instructions in the RUU 130. For example, Figure 5 shows 8 instructions numbered I1 through I8 in the representative RUU 130. Instruction I3 includes the value I1 in the DIN field which implies that the execution of I3 depends on the outcome of I1. Thus, execution units 134, 138, 142 recognize that instruction number I1 must be executed before instruction I3. Therefore, in the example shown in Figure 5, the same dependency exists between instructions I4 and I3 as well as I8 and I7. Meanwhile, independent instructions (*i.e.*, those with no number in the dependent instruction number field) may be executed out of order.

[0050] Referring still to Figure 4, the floating point register 122 and integer register 126 are used for the execution of instructions that require the use of such registers as is known by those of ordinary skill in the art. These registers 122, 126 can be loaded with data from the data cache 146. The registers also provide their contents to the RUU 130.

[0051] As shown, the execution units 134, 138, and 142 comprise a floating point execution unit 134, a load/store execution unit 138, and an integer execution unit 142. Each execution unit performs the operation specified by the corresponding instruction type. Accordingly, the floating

point execution units 134 execute floating instructions such as multiply and divide instruction while the integer execution units 142 execute integer-based instructions. The load/store units 138 perform load operations in which data from memory is loaded into a register 122 or 126. The load/ store units 138 also perform store operations in which data from registers 122, 126 is written to data cache 146 and/or DRAM memory 92 (Figure 1). The load/store units 138 also read the cycle counter 148 in response to read cycle count ("RCC") commands as they are encountered in a program thread. The function of the cycle count queue 150 is discussed in further detail below.

[0052] The architecture and components described herein are typical of microprocessors, and particularly pipelined, multithreaded processors. Numerous modifications can be made from that shown in Figure 4. For example, the locations of the RUU 130 and registers 122, 126 can be reversed if desired. For additional information, the following references, all of which are incorporated herein by reference, may be consulted for additional information if needed: U.S. Patent Application Serial No. 08/775,553, filed December 31, 1996, and "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreaded Processor," by D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo and R. Stamm, Proceedings of the 23rd Annual International Symposium on Computer Architecture, Philadelphia, PA, May 1996.

[0053] According to the preferred embodiment, the sphere of replication is represented by the dashed box shown in Figure 4. The majority of the pipelined processor components are included in the sphere of replication 200 with the notable exception of the instruction cache 110 and the data cache 146. The floating point and integer registers 122, 126 may alternatively reside outside of the sphere of replication 200, but for purposes of this discussion, they will remain as shown. The cycle counter clock 148 also resides outside of the sphere of replication and therefore, any reads from the cycle counter clock 148 must be replicated for the duplicate threads. Note also that the cycle count

queue 150 resides outside the sphere of replication as well. Thus, all information that is transmitted between the sphere of replication 200 and the cycle count queue 150 must be protected with some type of error detection, such as parity or error checking and correcting ("ECC"). Parity is an error detection method that is well-known to those skilled in the art. ECC goes one step further and provides a means of correcting errors. ECC uses extra bits to store an encrypted code with the data. When the data is written to a source location, the ECC code is simultaneously stored. Upon being read back, the stored ECC code is compared to the ECC code generated when the data was read. If the codes don't match, they are decrypted to determine which bit in the data is incorrect. The erroneous bit may then be flipped to correct the data.

[0054] The preferred embodiment provides an effective means of replicating cycle counter values returned from an RCC command in the leading thread T0 and delivering a "copy" to the trailing thread T1. Upon encountering an RCC command in the leading thread T0, the load/store units 138 load the current cycle count value from the cycle counter 148 as a conventional processor would. However, in addition, the preferred embodiment of the load/store units 138 loads the same cycle count value into the cycle count queue 150. The cycle count queue 150 is preferably a FIFO buffer that stores the cycle count values until the corresponding RCC commands are encountered in the trailing thread T1. The cycle count queue 150 preferably includes, at a minimum, the fields shown in Figure 6. Entries in the representative cycle count queue 150 shown in Figure 6 include an optional program count value and the cycle count value. The program count is used to properly identify the RCC instructions in the queue and the cycle count value is the value that was retrieved by the leading thread T0 when the RCC command was issued. The program count value field is optional because the FIFO buffer guarantees that cycle count values are retrieved by the trailing thread in the correct order. When an RCC command is issued in the trailing thread T1, the

load/store units 138 read the cycle count value from the cycle count queue 150 (and not the cycle counter 148). Since the buffer delivers the oldest cycle count values in the stack, and assuming the RCC commands are encountered in program order in the trailing thread, the same cycle count values are returned to each thread. The cycle count values are, therefore, properly replicated and erroneous faults are not generated. The assumed program order is maintained by creating appropriate dependencies in the RUU 130 (as discussed above) between the RCC commands and instructions immediately before or after the RCC command.

[0055] In order to prevent buffer overflow, it may be necessary to stall the leading thread T0 to permit the trailing thread T1 to access the cycle count queue 150 and therefore clear entries from the buffer. Similarly, if the queue becomes empty, it may be necessary, though unlikely, to stall the trailing thread T1 to allow cycle count values to enter the cycle count queue 150 before the trailing thread T1 accesses the queue.

[0056] In the event a cycle count queue 150 is unavailable or otherwise undesirable, an alternative embodiment exists whereby the leading thread T0 is stalled when an RCC command is encountered in the leading thread T0. In this alternative embodiment, the SRT processor 100 still comprises load/store units 138 and cycle counter 148, but the cycle count queue 150 is unnecessary. As the load/store units 138 encounter an RCC command in the leading thread, the execution of that command and all subsequent commands in the T0 thread is temporarily halted. SRT processor 100 fetches, executes, and retires instructions exclusively in the trailing thread T1 until the corresponding RCC command is encountered. At this point, the load/store units 138 will then execute the RCC command and return the cycle count value to both threads T0 and T1. It should be noted that if the SRT processor 100 is implementing slack fetch as described above, this feature must be temporarily disabled to permit synchronization of the threads. Naturally, disabling

the slack fetch feature will temporarily eliminate some of the advantages mentioned above, but this alternative embodiment permits implementation in older legacy systems that do not include a FIFO buffer that may be used as a cycle count queue 150. While this alternative embodiment is the less preferred of the two embodiments presented, it does permit implementation of transient fault detection in an existing computer system.

[0057] Accordingly, the preferred embodiment of the invention provides a method of replicating cycle counter values in an SRT processor that can execute the same instruction set in two different threads. The above discussion is meant to be illustrative of the principles and various embodiments of the present invention. Numerous variations and modifications will become apparent to those skilled in the art once the above disclosure is fully appreciated. It is intended that the following claims be interpreted to embrace all such variations and modifications.